Stable Leader Election
(Extended Abstract)

Marcos K. Aguilera[1], Carole Delporte-Gallet[2], Hugues Fauconnier[2], and Sam Toueg[3]

[1] Compaq Systems Research Center, 130 Lytton Ave, Palo Alto, CA, 94301, USA
Marcos.Aguilera@compaq.com
[2] LIAFA, Université D. Diderot, 2 Place Jussieu, 75251, Paris Cedex 05, France
{cd,hf}@liafa.jussieu.fr
[3] DIX, École Polytechnique, 91128 Palaiseau Cedex, France
sam@lix.polytechnique.fr

Abstract. We introduce the notion of *stable* leader election and derive several algorithms for this problem. Roughly speaking, a leader election algorithm is stable if it ensures that once a leader is elected, it remains the leader for as long as it does not crash and its links have been behaving well, irrespective of the behavior of other processes and links. In addition to being stable, our leader election algorithms have several desirable properties. In particular, they are all communication-efficient, i.e., they eventually use only n links to carry messages, and they are robust, i.e., they work in systems where only the links to/from some correct process are required to be eventually timely. Moreover, our best leader election algorithm tolerates message losses, and it ensures that a leader is elected in constant time when the system is stable. We conclude the paper by applying the above ideas to derive a robust and efficient algorithm for the eventually perfect failure detector $\Diamond P$.

1 Introduction

1.1 Motivation and Background

Failure detection is at the core of many fault-tolerant systems and algorithms, and the study of failure detectors has been the subject of intensive research in recent years. In particular, there is growing interest in developing failure detector implementations that are efficient, timely, and accurate [VRMMH98,LAF99,CTA00,FRT00,LFA00b].

A failure detector of particular interest is Ω [CHT96]. At every process p, and at each time t, the output of Ω at p is a single process, say q. We say that p *trusts* q *to be up at time* t. Ω ensures that eventually all correct processes trusts the *same* process and that this process is *correct*.

Note that a failure detector Ω can also be thought of as a leader elector: The process currently trusted to be up by p, can be thought of as the current "leader" of p, and Ω ensures that eventually all processes have the same leader.

An Ω leader election is useful in many settings in distributed systems. For example, some algorithms use it to solve consensus in asynchronous systems with failures [Lam98,MR00,LFA00a] (in fact, Ω is the weakest failure detector to solve consensus [CHT96]). Electing a leader can also be useful to solve a set of tasks efficiently

in distributed environments [DHW98]. Even though Ω is strong enough to solve hard problems such as consensus, we will see that it is weak enough to admit efficient implementations.

Our main goal here is to propose efficient algorithms for Ω in partially synchronous systems with process crashes and message losses. To illustrate this problem, consider the following simple implementation of Ω [Lam98,DPLL97]. Assume that processes may crash, but *all* links are eventually timely, i.e., there is a time after which all messages sent take at most δ time to be received. In this system, one can implement Ω as follows:

1. Every process p periodically sends an OK message to all, and maintains a set of processes from which it received an OK recently.[1]
2. The output of Ω at p is simply the smallest process currently in p's set.

Note that the set of processes that p builds in part (1) is eventually equal to the set of all correct processes. Thus part (1) actually implements an *eventually perfect failure detector* $\Diamond P$ [CT96]. So, in the above algorithm, we implement Ω by first implementing $\Diamond P$ and then outputting the smallest process in the set of processes trusted by $\Diamond P$.
This implementation of Ω has several drawbacks:

1. *The system assumptions required by the algorithm are too strong.* In fact, this algorithm works only if *all* links are eventually timely. Intuitively, however, one should be able to find a leader in systems where only the links to and from some correct process are eventually timely. In other words, while this algorithm requires n^2 eventually timely links, we would like an algorithm that works even if there are only n eventually timely links (those to and from some correct process).
2. *The algorithm is not communication-efficient.* In this algorithm, every process sends an OK message to all processes, forever. That is, all the n^2 links carry messages, in both directions, forever. Intuitively, this is wasteful: once a correct process is elected as a leader, it should be sufficient for it to periodically send OK messages to all processes (to inform them that it is still alive and so they can keep it as their leader), and all other processes can keep quiet. In other words, after an election is over, no more than n links should carry messages (those links from the elected leader to the other processes). All the other links should become quiescent. We say that a leader election algorithm is *communication-efficient* if there is a time after which it uses only unidirectional n links.
3. *The election is not stable.* In this algorithm, processes can demote their current leader and elect a new leader for no real reason: even if the current leader has not crashed and its links have been timely for an (arbitrarily) long time, the leader can still be demoted at any moment by an extraneous event. To see this, suppose process 2 is trusted forever by $\Diamond P$ (because it is correct and all its links are timely) and that it is the current leader (because it is the smallest process currently trusted by $\Diamond P$). If $\Diamond P$ starts trusting process 1 (this can occur if the links from process 1 become

[1] "Recent" means within Δ from the last OK received. If processes send OK every η then $\Delta = \delta + \eta$. If δ and η are not known, p sets Δ by incrementing it for every mistake it makes [CT96].

timely), then 2 loses the leadership and 1 is elected. If later 1 is suspected again, 2 regains the leadership. So 2 loses the leadership each time 1 becomes "good" again, *even though 2 keeps behaving well and remains trusted forever by all the processes!* This is a serious drawback, because leadership changes are quite costly to most applications that rely on a leader. Thus, we are seeking a *stable* leader election algorithm. Roughly speaking, such an algorithm ensures that once a leader is elected, it remains the leader for as long as it does not crash and its links have been behaving well, irrespective of the behavior of other processes or links.

Our main goal here is to give algorithms for Ω (i.e., leader election algorithms) that are communication-efficient, stable, and that work in systems where only the links to and from some correct process are required to be eventually timely, as explained above. In addition, we want an algorithm for Ω that can elect a leader quickly when the system "stabilizes", i.e., it has a small *election time*.

We achieve our goal progressively. We first present an algorithm for Ω that is communication-efficient. This algorithm is simple, however it has the following drawbacks: (a) it is not stable, (b) it assumes that messages are not lost and (c) its worst-case election time is proportional to n,[2] even when the system is stable. We next modify this algorithm to achieve stability. Then we change it so that it works despite message losses. Finally, we modify it to achieve constant election time when the system "stabilizes". It is worth noting that our algorithms are self-stabilizing.

We conclude the paper by using our techniques to give an algorithm for $\Diamond P$ that is both robust and efficient: In contrast to previous implementations of $\Diamond P$, our algorithm works in systems where only n bidirectional links are required to be eventually timely, and there is a time after which only n bidirectional links carry messages. This algorithm for $\Diamond P$ works despite message losses.

1.2 Related Work

The simple implementation of Ω described above is mentioned in several works (e.g., [Lam98,DPLL97]). Such an implementation, however, requires strong systems assumptions, is not communication-efficient, and is not stable. Larrea et al. give an algorithm for Ω that is communication-efficient, but it requires strong systems assumptions, and is not stable [LFA00b]. An indirect way to implement Ω is to first implement an eventually strong failure detector $\Diamond S$ [CT96] and then transform it into Ω using the algorithms in [Chu98]. But such implementations also have drawbacks. First, the known implementations of $\Diamond S$ are either not communication-efficient [CT96,ACT99,ACT00] or they require strong system assumptions [LAF99,LFA00b]. Second, the Ω that we get this way is not necessarily stable.

To the best of our knowledge, all prior implementations of $\Diamond P$ require that $O(n^2)$ links to be eventually timely. Larrea et al propose a communication-efficient transformation of Ω to $\Diamond P$, but it requires all links to be eventually timely and it does not tolerate message losses [Lar00].

[2] Actually, it is proportional to the maximum number of failures.

1.3 Summary of Contributions

The contributions of the paper are the following:

- We introduce the notion of *stable* leader election and describe the first leader election algorithm that is simultaneously stable and communication-efficient and requires only n eventually timely bidirectional links.
- We modify our algorithm to work with message losses, first when processes have approximately synchronized clocks, and then when clocks are drift-free or have bounded drift.
- We show how to achieve constant election time during good system periods.
- We give an algorithm for $\Diamond P$ that is both robust and efficient: it works in systems where only n bidirectional links are required to be eventually timely, and there is a time after which only n bidirectional links carry messages.

1.4 Roadmap

This paper is organized as follows. In Sect. 2 we describe our model. In Sect. 3, we define the problem of stable and message-efficient Ω leader election. In Sect. 4, we give a simple algorithm for Ω, and then modify it in Sect. 5 to make it stable. In Sect. 6, we give algorithms for Ω that work despite message losses. Then, in Sect. 7, we explain how to obtain Ω with a constant election time when the system stabilizes. In Sect. 8 we discuss view numbers. Finally, in Sect. 9, we give a new algorithm for $\Diamond P$ that guarantees that, there is a time after which, only n bidirectional links carry messages.

Because of space limitations in this extended abstract, we have omitted all technical proofs. They can be found in [ADGFT01].

2 Informal Model

We consider a distributed system with $n \geq 2$ processes $\Pi = \{0, \ldots, n-1\}$ that can communicate with each other by sending messages through a set of links Λ. We assume that the network is fully connected, i.e., $\Lambda = \Pi \times \Pi$. The link from process p to process q is denoted by $p \longrightarrow q$. The system is partially synchronous in that (1) links are sometimes timely (good) and sometimes slow, (2) processes have drift-free clocks (which may or may not be synchronized) and (3) there is an upper bound B on the time a process takes to execute a step. For simplicity, we assume that $B = 0$, i.e., processes execute a step instantaneously, but it is easy to modify our results for any $B \geq 0$. At each step a process can (1) receive a message, (2) change its state and (3) send a message. The value of a variable of a process at time t is the value of that variable after the process takes a step at time t.

Processes and process failure patterns. Processes can fail by crashing, and crashes are permanent. A process failure pattern is function F_P that indicates, for each time t, what processes have crashed by t. We say that *process p is alive at time t* if $p \notin F_P(t)$. We say process p is correct if it is always alive.

Link behavior pattern. A link behavior pattern is a function F_L that determines, for each time t, which links are good at t. The guarantees provided by links when they

are good are specified by axiomatic links properties. These properties, given below, depend on whether the link is reliable or lossy.

Reliable links. Some of our basic algorithms require reliable links. Such links do not create, duplicate or drop messages. The link may sometimes be good and sometimes slow. If a process sends a message through a link and the link remains good for δ (δ is a system parameter known by processes) then the recipient gets the message within δ. More precisely, a reliable link $p \longrightarrow q \in \Lambda$ satisfies the following properties:

- *(No creation or duplication):* Process q receives a message m from p at most once, and only if p previously sent m to q.
- *(No late delivery in good periods):* If p sends m to q by time $t - \delta$ and $p \longrightarrow q$ is good during times $[t - \delta, t]$ then q does not receive m after time t.
- *(No loss):* If p sends m to q then q eventually receives m from p.[3]

Lossy links. Like reliable links, lossy links do not create or duplicate messages and may be slow or not. However, unlike reliable links, they may drop messages when they are not good. A lossy link $p \longrightarrow q \in \Lambda$ satisfies the following properties:

- *(No creation or duplication):* Same as above.
- *(No late delivery in good periods):* Same as above.
- *(No loss in good periods):* If p sends m to q at time $t - \delta$ and $p \longrightarrow q$ is good during times $[t - \delta, t]$ then q eventually receives m from p.

Connectivity. In this paper, we focus on implementing Ω (defined below). It is easy to show that this is impossible if links are never good. We thus assume that there exists at least one process whose links are eventually good. More precisely, we say that a process p is *accessible at time* t if it is alive at time t and all links to and from p are good at time t.[4] We say that p *is eventually accessible* if there exists a time t such that p is accessible at every time after t.[5] We assume that there exists at least *one* process that is eventually accessible.

3 Stable Leader Election

3.1 Specification of Ω

We consider a weak form of leader election, denoted Ω, in which each process p has a variable $leader_p$ that holds the identity of a process or $\perp$.[6] Intuitively, eventually all alive processes should hold the identity of the same process, and that process should be correct. More precisely, we require the following property:

[3] For convenience, in our model dead processes "receive" messages that are sent to them (but of course they cannot process such messages).

[4] For convenience, we assume that a process is not accessible at times $t < 0$.

[5] Note that eventually-forever accessible would be a more precise name for this property, but it is rather cumbersome.

[6] The original definition of Ω does not allow the output to be $\perp$. We allow it here because it is convenient for processes to know when the leader elector has not yet selected a leader.

– There exists a correct process ℓ and a time after which, for every alive process p, $leader_p = \ell$.

If at time t, *leader*$_p$ contains the same process ℓ for all alive processes p, then we say that ℓ *is leader at time* t. Note that a process p never knows if *leader*$_p$ is really the leader at time t, or not. A process only knows that *eventually leader*$_p$ is leader. This guarantee seems rather weak, but it is actually quite powerful: it can be used to solve consensus in asynchronous systems [CHT96].

3.2 Communication-efficiency

An algorithm for Ω is communication-efficient if there is a time after which it uses only n unidirectional links. All our Ω algorithms are communication-efficient. Actually, if we discount messages from a process to itself, our algorithms use only $n - 1$ links, which is optimal [LFA00b].

3.3 Stability

A change of leadership often incurs overhead to an application, which must react to deal with the new leader. Thus, we would like to avoid switching leaders as much as possible, unless there is a good reason to do so. For instance, if the leader has died or has been inaccessible to processes, it must be replaced. An algorithm that changes leader *only* in those circumstances is called *stable*. More precisely, a k-stable algorithm guarantees that in every run,

– if p is leader at time t and p is accessible during times $[t - k\delta, t + 1]$ then p is leader at time $t + 1$.

Here, k is a parameter that depends on the algorithm; the smaller the k, the better the algorithm because it provides a stronger stability property. We introduced parameter k because no algorithm can be "instantaneously" stable (0-stable) and 1-stable algorithms have serious drawbacks, as we show in the full paper [ADGFT01]. Our algorithm for reliable links is 3-stable while our best algorithm for lossy links is 6-stable.

4 Basic Algorithm for Ω

Figure 1 shows an algorithm for Ω that works in systems with reliable links. This algorithm is simple and communication-efficient but not stable — we will later modify it to get stability. Intuitively, processes execute in rounds $r = 0, 1, 2, \ldots$, where variable r keeps the process's current round. To start a round k, a process (1) sends $(START, k)$ to a specially designated process, called the "leader of round k"; this is just process $k \bmod n$, (2) sets r to k, (3) sets the output of Ω to $k \bmod n$ and (4) starts a timer — a variable that is automatically incremented at each clock tick. While in round r, the process checks if it is the leader of that round (task 0) and if so sends (OK, r) to

```
Code for each process p:
1   procedure StartRound(s)                          { executed upon start of a new round }
2     if p ≠ s mod n then send (START, s) to s mod n   { wake up new leader }
3     r ← s                                           { update current round }
4     leader ← s mod n                                { output of Ω }
5     restart timer

6   on initialization:
7     StartRound(0)
8     start tasks 0 and 1

9   task 0:                                           { leader sends OK every δ time }
10    loop forever
11      if p = r mod n and have not sent (OK, r) within δ then send (OK, r) to all

12  task 1:
13    upon receive (OK, k) with k = r do              { current leader is active }
14      restart timer

15    upon timer > 2δ do                              { timeout on current leader }
16      StartRound(r + 1)                             { start next round }

17    upon receive (OK, k) or (START, k) with k > r do
18      StartRound(k)                                 { jump to round k }
```

Fig. 1. Basic algorithm for Ω with reliable links.

all every δ time.[7] When a process receives an (OK, k) for the current round ($r = k$), the process restarts its timer. If the process does not receive (OK, r) for more than 2δ time, it times out on round r and starts round $r + 1$. If a process receives (OK, k) or $(START, k)$ from a higher round ($k > r$), the process starts that round.

Intuitively, this algorithm works because it guarantees that (1) if the leader of the current round crashes then the process starts a new round and (2) processes eventually reach a round whose leader is a correct process that sends timely (OK, k) messages.

Theorem 1. *Consider a system with reliable links. Assume some process is eventually accessible. Figure 1 is a communication-efficient algorithm for Ω.*

5 Stable Algorithm for Ω

The algorithm of Fig. 1 implements Ω but it is not stable because it is possible that (1) some process q is accessible for an arbitrarily long time, (2) all alive processes have q

[7] In this and other algorithms, we chose the sending period to be equal to the network delay δ. This arbitrary choice was made for simplicity of presentation only. In general, the sending period can be set to any value η, though, in this case, one needs to modify the algorithms slightly, e.g., by adjusting the time out periods. The choice of η affects the quality of service of the failure detector [CTA00], such as how fast a leader is demoted if it crashes.

as their leader at time t, but (3) q is demoted at time $t+1$. This could happen in two essentially different ways:

Problem scenario 1. Initially all processes are in round 0 and so process 0 is the leader. All links are good (timely), except the links to and from process 2, which are very slow. Then at time $2\delta+1$, process 2 times out on round 0 and starts round 1, and so 0 loses leadership. Moreover, 2 sends $(START, 1)$ to process 1. At time $2\delta+2$, process 2 crashes and process 0 becomes accessible. Message $(START, 1)$ is delayed until some arbitrarily large time $M \gg 2\delta+2$. At time M, process 1 receives $(START, 1)$ and starts round 1, and thus process 0 is no longer leader.

In this scenario, process 0 becomes the leader right after process 2 crashes, since all alive processes are then in round 0 and hence have 0 as their leader. Unfortunately 0 is demoted at time M, even though it has been accessible to all processes during the arbitrarily long period $[2\delta+2, M]$.

Problem scenario 2. We divide this scenario in two stages. *(A) Setup stage.* Initially process 1 times out on round 0, starts round 1 and sends $(OK, 1)$ to all. All processes except process 0 get this message and start round 1. Then process 3 times out on rounds 1 and 2, starts round 3 and sends $(OK, 3)$ to all. All processes except process 0 get that message and start round 3; process 0, however, remains in round 0 (because all messages from higher rounds are delayed). Then process 2 becomes accessible. All processes except 0 remain in round 3 for a long time, while 0 remains in round 0 for a long time. All processes except 0 then progressively time out on rounds $3, 4, \ldots$ until they start round $n+2$, say at time t. Meanwhile, process 0 receives the old $(OK, 1)$ message, advances to round 1, timeouts on round 1 and starts round 2 at time t. *(B) Demote stage.* Note that at time t, process 2 is the leader because all processes are in a round congruent to 2 modulo n. Moreover, 2 has been accessible for a long time. Unfortunately, process 2 stops being the leader when process 0 receives $(OK, 3)$ and starts round 3.

Summary of bad scenarios. Essentially, scenario 1 is problematic because a single process may (1) time out on the current round, (2) send a message to move to a higher round and then (3) die. This message may be delayed and may demote the leader long in the future. On the other hand, scenario 2 is problematic because a process may become a leader while processes are in different rounds; after the leader is elected, a process in a lower round may switch its leader by moving to a higher round.

Our new algorithm, shown in Fig. 2, avoids the above problems. To prevent problem scenario 1, when a process times out on round k, it sends a $(STOP, k)$ message to $k \bmod n$ before starting the next round. When $k \bmod n$ receives such a message, it abandons round k and starts round $k+1$. To see why this prevents scenario 1, note that, before process 2 sends $(START, 1)$ to 1, it sends $(STOP, 0)$ to 0. Soon after process 0 becomes accessible, it receives such a message and abandons round 0.

To avoid problem scenario 2, when a process starts round k, it no longer sets *leader* to $k \bmod n$. Instead, it sets it to $\perp$ and waits until it receives two (OK, k) messages from $k \bmod n$. Only then it sets leader to $k \bmod n$. This guarantees that if $k \bmod n$ is accessible and some process sets $leader$ to $k \bmod n$, then all processes have received at least one (OK, k) and hence have started round k. In this way, all processes are in the same round k.

```
   Code for each process p:
1    procedure StartRound(s)                          { executed upon start of a new round }
2      if p ≠ s mod n
3      then send (START, s) to s mod n                { wake up the new leader candidate }
4      r ← s                                                          { update current round }
5      leader ← ⊥          { demote previous leader but do not elect leader quite yet }
6      restart timer
7    on initialization:
8      StartRound(0)
9      start tasks 0 and 1
10   task 0:                                  { leader/candidate sends OK every δ time }
11     loop forever
12       if p = r mod n and have not sent (OK, r) within δ then send (OK, r) to all
13   task 1:
14     upon receive (OK, k) with k = r do             { current leader/candidate is active }
15       if leader = ⊥ and received at least two (OK, k) messages
16       then leader ← k mod n                                        { now elect leader }
17       restart timer
18     upon timer > 2δ do                          { timeout on current leader/candidate }
19       send (STOP, r) to r mod n                    { stop current leader/candidate }
20       StartRound(r + 1)                                            { start next round }
21     upon receive (STOP, k) with k ≥ r do     { current leader abdicates leadership }
22       StartRound(k + 1)                                            { start next round }
23     upon receive (OK, k) or (START, k) with k > r do
24       StartRound(k)                                                { jump to round k }
```

Fig. 2. 3-stable algorithm for Ω with reliable links.

Theorem 2. *Consider a system with reliable links. Assume some process is eventually accessible. Figure 2 is a 3-stable communication-efficient algorithm for Ω.*

6 Stable Ω with Message Losses

In our previous algorithm, we assumed that links do not lose messages, but in many systems this is not the case. We now modify our algorithms to deal with message losses. First note that if *all* messages can be lost, there is not much we can do, so we assume there is at least one eventually accessible process p. That means that p is correct and there is a time after which the links to and from p do not drop messages and are timely (see Sect. 2).

6.1 Expiring Links

So far, our model allowed links to deliver messages that have been sent long in the past. This behavior is undesirable because an out-of-date message can demote a leader that has been good recently. To solve this problem, we now use links that discard messages older than δ — we call them *expiring links*. Such links can be easily implemented from "plain" lossy links if processes have approximately synchronized, drift-free clocks, by using timestamps to expire old messages. We now make these ideas more precise.

Informal definition. Expiring links are lossy links that automatically drop old messages. To model such links, we change the property "No late delivery in good periods" of lossy links (Sect. 2) to require no late delivery *at all times*, not just when the link is good. More precisely, an expiring link $p \longrightarrow q$ satisfies the following properties:

- *(No late delivery):* If p sends m to q by time $t - \delta$ then q does not receive m after t.
- *(No creation or duplication):* (same as before) Process q receives a message m from p at most once, and only if p previously sent m to q.
- *(No loss in good periods):* (same as before) If p sends m to q at time $t - \delta$ and $p \longrightarrow q$ is good during times $[t - \delta, t]$ then q eventually receives m from p.

Implementation. If processes have perfectly synchronized clocks, we can easily implement expiring links from plain lossy links as follows: (1) the sender timestamps m before sending it and (2) the receiver checks the timestamp and discards messages older than δ. This idea also works when processes have ϵ-synchronized, drift-free clocks, though the resulting link will have a δ parameter that is 2ϵ larger than the δ of the original links. It is also possible expire messages even if clocks are not synchronized, provided they are drift-free or have a bounded drift, as we show in the full paper [ADGFT01].

Henceforth, we assume that all links are expiring links (this holds for all our Ω algorithms that tolerate message losses).

6.2 $O(n)$-Stable Ω

Figure 3 shows an $(n + 4)$-stable algorithm for Ω that works despite message losses. The algorithm is similar to our previous algorithm that assumes reliable links (Fig. 2), with only three differences: the first one is that in line 2, p sends the $START$ message to all processes, not just to $s \bmod n$. The second difference is that there are no *STOP* messages. And the last difference is the addition of lines 21 and 22; without these two lines, the algorithm would not implement Ω (it is not hard to construct a scenario in which the algorithm fails).

This new algorithm is $O(n)$-stable rather than $O(1)$-stable. To see why, consider the following scenario. Initially all processes are in round 0. At time $2\delta + 1$ the following happens: (1) process 2 times out on round 0 and attempts to sends $(START, 1)$ to all, but crashes during the send and only sends to process 3 and (2) process 0 becomes accessible. At time $3\delta + 1$, process 3 receives $(START, 1)$ and tries to send $(START, 1)$ to all, but crashes and only sends to process 4. And so on. Then at time $n\delta + 1$, process 0 is the leader but it receives $(START, 1)$ from process $n - 1$ and demotes itself, even though it has been accessible during $[2\delta + 1, n\delta + 1]$.

```
Code for each process p:
1   procedure StartRound(s)                          { executed upon start of a new round }
2     if p ≠ s mod n then send (START, s) to all      { bring all to new round }
3     r ← s                                          { update current round }
4     leader ← ⊥          { demote previous leader but do not elect leader quite yet }
5     restart timer

6   on initialization:
7     StartRound(0)
8     start tasks 0 and 1

9   task 0:                                  { leader/candidate sends OK every δ time }
10    loop forever
11      if p = r mod n and have not sent (OK, r) within δ then send (OK, r) to all

12  task 1:
13    upon receive (OK, k) with k = r do     { current leader/candidate is active }
14      if leader = ⊥ and received at least two (OK, k) messages
15      then leader ← k mod n                                  { now elect leader }
16      restart timer

17    upon timer > 2δ do                     { timeout on current leader/candidate }
18      StartRound(r + 1)                                      { start next round }

19    upon receive (OK, k) or (START, k) with k > r do
20      StartRound(k)                                          { jump to round k }

21    upon receive (OK, k) or (START, k) from q with k < r do
22      send (START, r) to q                       { update process in old round }
```

Fig. 3. $(n + 4)$-stable algorithm for Ω that tolerates message losses.

Theorem 3. *Consider a system with message losses (expiring links). Assume some process is eventually accessible. Figure 3 is an* $(n + 4)$*-stable communication-efficient algorithm for* Ω*.*

6.3 $O(1)$-Stable Ω

Our previous algorithm can tolerate message losses but it is only $O(n)$-stable. This can be troublesome if the number n of processes is large. We now provide a better algorithm that is 6-stable. We manage to get $O(1)$-stability by ensuring that a leader is not elected if there are long chains of messages that can demote the leader in the future.

Our algorithm is shown in Fig. 4. It is identical to our previous algorithm, except that there is a new $(ALERT, k)$ message. This message is sent to all when a process starts round k. When a process receives such a message from a higher round, the process *temporarily* sets its leader variable to $\perp$ for 6δ time units. However, unlike with a *START* message, the process does not advance to the higher round.

```
Code for each process p:
1   procedure StartRound(s)                          { executed upon start of a new round }
2     send (ALERT, s) to all
3     if p ≠ s mod n then send (START, s) to all      { bring all to new round }
4     r ← s                                          { update current round }
5     leader ← ⊥          { demote previous leader but do not elect leader quite yet }
6     restart timer
7   on initialization:
8     StartRound(0)
9     start tasks 0 and 1
10  task 0:                                  { leader/candidate sends OK every δ time }
11    loop forever
12      if p = r mod n and have not sent (OK, r) within δ then send (OK, r) to all
13  task 1:
14    upon receive (OK, k) with k = r do     { current leader/candidate is active }
15      if leader = ⊥ and received at least two (OK, k) messages
16        and did not receive (ALERT, k') with k' > k within 6δ
17      then leader ← k mod n                                  { now elect leader }
18      restart timer
19    upon timer > 2δ do                     { timeout on current leader/candidate }
20      StartRound(r + 1)                                      { start next round }
21    upon receive (OK, k) or (START, k) with k > r do
22      StartRound(k)                                          { jump to round k }
23    upon receive (OK, k) or (START, k) from q with k < r do
24      send (START, r) to q                       { update process in old round }
25    upon receive (ALERT, k) with k > r do
26      leader ← ⊥                                             { suspend current leader }
```

Fig. 4. 6-stable algorithm for Ω that tolerates message losses.

Theorem 4. *Consider a system with message losses (expiring links). Assume some process is eventually accessible. Figure 4 is a* 6*-stable communication-efficient algorithm for* Ω*.*

7 Stable Ω with Constant Election Time

In some applications, it is important to have a small *election time* — the time to elect a new leader when the system is leaderless. This time is inevitably large if there are crashes or slow links during the election. For instance, if an about-to-be leader crashes right before being elected, the election has to start over and the system will continue to be leaderless. Slow links often cause the same effect.

```
20      send (ALERT, r + 1) to all
20.1    send (PING, r) to all                                   { ask who is alive }
20.2    wait for 2δ time or until receive (OK, k) or (START, k) with k > r
20.3    if received (OK, k) or (START, k) with k > r then return
20.4    S ← {q: received (PONG, r) from q }                     { responsive processes }
                  { we assume that p responds to itself immediately, so S is never ∅ }
20.5    k ← smallest k' > r such that k' mod n ∈ S   { round of next responsive process }
20.6    StartRound(k)                                           { start next round }

20.7  upon receive (PING, k) from q do
20.8    send (PONG, k) to q
```

Fig. 5. Improving the election time in the algorithm of Fig. 4.

It is possible, however, to ensure small election time during *good periods* — periods with no slow links or additional crashes. In such periods, the election time of our previous algorithms is proportional to f, the number of crashes so far. This is because processes may go through f rounds trying to elect processes who are long dead. With a simple modification, however, it is possible to do much better and achieve a constant election time (independent of f). The basic idea is that, when a process wants to start a new round, it first queries all processes to determine who is alive. Then, instead of starting the next round, the process skips all rounds of unresponsive processes. Using this idea, we can get a 6-stable algorithm with an election time of 9δ, as follows: we take the algorithm of Fig. 4 and replace its line 20 with the code shown in Fig. 5. [8]

Theorem 5. *Consider a system with message losses (expiring links). Assume some process is eventually accessible. If we replace line 20 in Fig. 4 with the code in Fig. 5 we obtain a* 6*-stable communication-efficient algorithm for* Ω*. Its election time is* 9δ *when there are no slow links or additional crashes.*

8 Leader Election with View Numbers

It may be useful to tag leaders with a *view number* such that there is at most one leader per view number and eventually processes agree on the view number of the leader. More precisely, we define a variant of Ω, which we call Ω^+, in which each process outputs a pair (p, v) or $\perp$, where p is a process and v is a number. Ω^+ guarantees that (1) if some process outputs (p, v) and some process outputs (q, v) then $p = q$ and (2) there exists a correct process ℓ, a number v_ℓ and a time after which, for every alive process p, p outputs (ℓ, v_ℓ).

It turns out that our Ω algorithms can be made to output a view number with no modifications: they can simply output the current round r. By doing that, it is not hard to verify that our algorithms actually implement Ω^+.

[8] The same idea can be applied to get a constant election with our other algorithms.

9 An Efficient Algorithm for $\Diamond P$

Recall that, at each alive process p, the eventually perfect failure detector $\Diamond P$ outputs a set of trusted processes, such that there is a time after which the set of p contains process q if and only if q is correct. We now give an algorithm for $\Diamond P$ that is both robust and efficient: In contrast to previous implementations of $\Diamond P$, our algorithm works in systems where only n bidirectional links are required to be eventually timely, and there is a time after which only n bidirectional links carry messages.

Our algorithm, shown in Fig. 6, tolerates message losses with expiring links. It is based on the algorithm in Fig. 3, and the difference is that (1) there are no *ALERT* messages and (2) there is a mechanism to get the list *trust* of trusted processes of $\Diamond P$: When processes receive *OK*, they send *ACK* to the leader, and the leader sets *trust* to the set of processes from which it received *ACK* recently. The leader then sends its *trust* to other processes, by piggybacking it in the *OK* messages. Upon receiving *OK*, a process q checks if the leader's *trust* contains q. If so, the process sets its own *trust* to the leader's. Else, the process notices that the leader has made a mistake, and so it starts the next round.

We assume that if a process sends a message to itself, that message is received and processed immediately.

Theorem 6. *Consider a system with message losses (lossy links). Assume some process is eventually accessible. Figure 6 is an algorithm for $\Diamond P$. With this algorithm, there is a time after which only n bidirectional links carry messages.*

References

[ACT99] M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, June 1999.

[ACT00] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, April 2000.

[ADGFT01] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. Technical Report 2001/04, LIAFA, Université D. Diderot, 2 Place Jussieu, 75251, Paris Cedex 05, France, 2001.

[CHT96] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[Chu98] F. Chu. Reducing Ω to $\diamond W$. *Information Processing Letters*, 67(6):298–293, September 1998.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[CTA00] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. In *International Conference on Dependable Systems and Networks (DSN 2000)*, pages 191–200, New York, June 2000. A full version of this paper will appear in the IEEE Transactions on Computers.

[DHW98] C. Dwork, J.Y. Halpern, and O. Waarts. Performing work efficiently in the presence of faults. *SIAM Journal on Computing*, 27(5):1457–1491, 1998.

[DPLL97] R. De Prisco, B. Lampson, and N. Lynch. Revisiting the Paxos algorithm. In *Proceedings of the 11th Workshop on Distributed Algorithms(WDAG)*, pages 11–125, Saarbrücken, September 1997.

```
   Code for each process p:
1    procedure StartRound(s)                                   { executed upon start of a new round }
2      if p ≠ s mod n then send (START, s) to all
3      r ← s                                                   { update current round }
4      leader ← s mod n
5      trust ← Π                                               { trust all initially }
6      restart timer

7    on initialization:
8      StartRound(0)
9      start tasks 0 and 1

10   task 0:                                                   { leader updates trust and sends OK every δ time }
11     loop forever
12       if p = r mod n and have not sent (OK, r, ∗) within δ then
13         if p has been in round r for at least 2δ time then
14           for each q ∈ trust s.t. p did not receive (ACK, r) from q in the last 2δ time do trust ← trust \ {q}
15         send (OK, r, trust) to all

16   task 1:
17     upon receive (OK, k, tr) with k = r do                  { current leader is active }
18       if p ∉ tr then StartRound(r + 1)                      { leader does not trust p, so p starts new round }
19       else
20         trust ← tr
21         send (ACK, r) to r mod n
22         restart timer

23     upon timer > 2δ do                                      { timeout on current leader }
24       StartRound(r + 1)                                     { start next round }

25     upon receive (OK, k, tr) or (START, k) with k > r do
26       if received (OK, ∗, ∗) then send (ACK, k) to k mod n
27       StartRound(k)                                         { jump to round k }

28     upon receive (OK, k, tr) or (START, k) from q with k < r do
29       send (START, r) to q
```

Fig. 6. An efficient algorithm for $\Diamond P$.

[FRT00] C. Fetzer, M. Raynal, and F. Tronel. A failure detection protocol based on a lazy approach. Research Report 1367, IRISA, November 2000.

[LAF99] M. Larrea, S. Arévalo, and A. Fernández. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In *Proceedings of the 13th International Symposium on Distributed Algorithms(DISC99)*, pages 34–48, Bratislava, September 1999.

[Lam98] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[Lar00] M. Larrea, November 2000. Personal communication.

[LFA00a] M. Larrea, A. Fernández, and S. Arévalo. Eventually consistent failure detectors. In *Brief Annoucement the 14th International Symposium on Distributed Algorithms(DISC00)*, Toledo, October 2000.

[LFA00b] M. Larrea, A. Fernández, and S. Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *in Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems, SRDS 2000*, pages 52–59, Nurenberg, Germany, October 2000.

[MR00] A. Mostefaoui and M. Raynal. Leader-based consensus. Research Report 1372, IRISA, December 2000.

[VRMMH98] R. Van Renesse, Y. Minsky, and M. M. Hayden. A gossip-based failure detection service. In *Proceedings of Middleware '98 (Sept. 1998)*, September 1998.